Announcements

- PA2 grades are released
- Quiz 6 will be next Tuesday
- Homework 3 is released, due next Thursday at 3:30pm
- Programming Project 3 is ongoing, due today (at 11:59pm)

Topic 6: Searching Linear Structures

By: Professor Lynam

Sequential (Linear) Search

- Scans each element in order of index to find a specific element. (Typically returns false or none/null upon failure)
- Best Case: O(1) – if the target is at list[0]
- Worst Case: O(n) – if the target is at list[len-1] or if the target is not found
- Average Case (assuming only successful searches): when all targets are equally likely. Let's sum up the number of comparisons we have to make at each index.
- If the target is at index 1, just 1 comparison, if at 2, then 2 comparisons… if at n, then n comparisons

Sequential (Linear) Search

- To the find the average of all of these cases, since each is equally likely, and there are n cases to consider, we can just add them all up and divide by n:
- $\frac{1+2+\cdots+n}{n} = \frac{\sum_1^n i}{n}$
- $\frac{\sum_1^n i}{n} = \frac{\frac{n*(n+1)}{2}}{n} = \frac{n+1}{2}$ which is Big-O of…?
- O(n)
- Which is obvious, but it is good to establish what we mean by the “average” search case

Binary Search

- Binary search needs, ordered data and a direct-access data structure

0	1	2	3	4	5	6	7	8	9	10
2	6	7	14	19	21	36	42	61	77	81

- Target: 14

Probe	Low	High	Mid	Key Comparison
1	0	10	5	21 > 14; eliminate top portion
2	0	4	2	7 < 14; eliminate bottom portion
3	3	4	3	14 = 14; return success

Binary Search

- Binary search needs, ordered data and a direct-access data structure

0	1	2	3	4	5	6	7	8	9	10
2	6	7	14	19	21	36	42	61	77	81

- Target: 63

Probe	Low	High	Mid	Key Comparison
1	0	10	5	21 < 63; eliminate bottom portion
2	6	10	8	61 < 63; eliminate bottom portion
3	9	10	9	77 > 63; eliminate top portion
4	9	8	-	High and low crossed; return failure

Binary Search

- Binary search efficiency:

# keys examined:	1	2	3	…	k	$\log_2 x + 1$
# keys in list:	1	3	7	…	$2^k - 1$	x

- A different perspective… # of probes per index. Let n = 15. Let's see how many probes it takes to find any given value in the list.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
4	3	4	2	4	3	4	1	4	3	4	2	4	3	4

- So, assuming all keys are equally likely to be searched, over 50% of the time we're performing the maximum number of probes!

Binary Search

- Given those perspectives, the analysis is straight-forward:
- Best Case: O(1), when the target is at the midpoint
- Worst Case: O($\log_2 n$), for the maximum number of probes
- Average Case: O($\log_2 n$), since ½ of the possible targets are the worst-case

Binary Search

- Binary search is pretty efficient! Can it be improved though?
- Yes, *if* the data cooperates. We'll need:

1. Binary Search's preconditions (ordered data, direct-access structure)
2. A *lot* of data to be searched
3. Data that is uniformly-distributed (no clusters or gaps)

- Those additional conditions allow us to interpolate a target's likely location within the data, leading to…

Interpolation Search

- Suppose we need to find 780 with the array below:

	0 (low)	1	2		77		99 (high)
List:	10	20	30	…	780	…	1000

- 780 is $\frac{780-10}{1000-10} = \frac{7}{9}$ths into the data's range (10 to 1000).
- Let's guess that 780's location is $\frac{7}{9}$ths into the range of indices, starting from the low index of that range:
- probe = $low + \left\lfloor \frac{target - list[low]}{list[hi] - list[low]} * (high - low) \right\rfloor$
- $0 + \left\lfloor \frac{780-10}{1000-10} * (99 - 0) \right\rfloor = 77$

Interpolation Search

- What about when the data's distribution isn't *perfectly* uniform?
- For example, suppose we have list[0…9999], keys 5…95,000, and target is 85,000. What would our first probe be?
- First probe: $0 + \left\lfloor \frac{85000-5}{95000-5} * (9999 - 0) \right\rfloor = 8946$
- Suppose we find 86,500 at list[8946]. 85,000 is smaller, so we just slide the 'high' down to index 8945.
- Assuming list[8945]=86487, what would our second probe be?
- $0 + \left\lfloor \frac{85000-5}{86487-5} * (8945 - 0) \right\rfloor = 8791$
- Just continue the search like this until success or failure!

Interpolation Search

- Interpolation search is just binary search with a different formula for selecting probe indices… but the efficiency analysis is not the same
- Best Case: O(1), when the target is at list[probe]
- Worst Case: O(n), when the data is badly distributed!
- Average Case: O($\log_2(\log_2 n)$); it's a very complex analysis
- Is interpolation search even worth it, given the worst case is the same as sequential search?

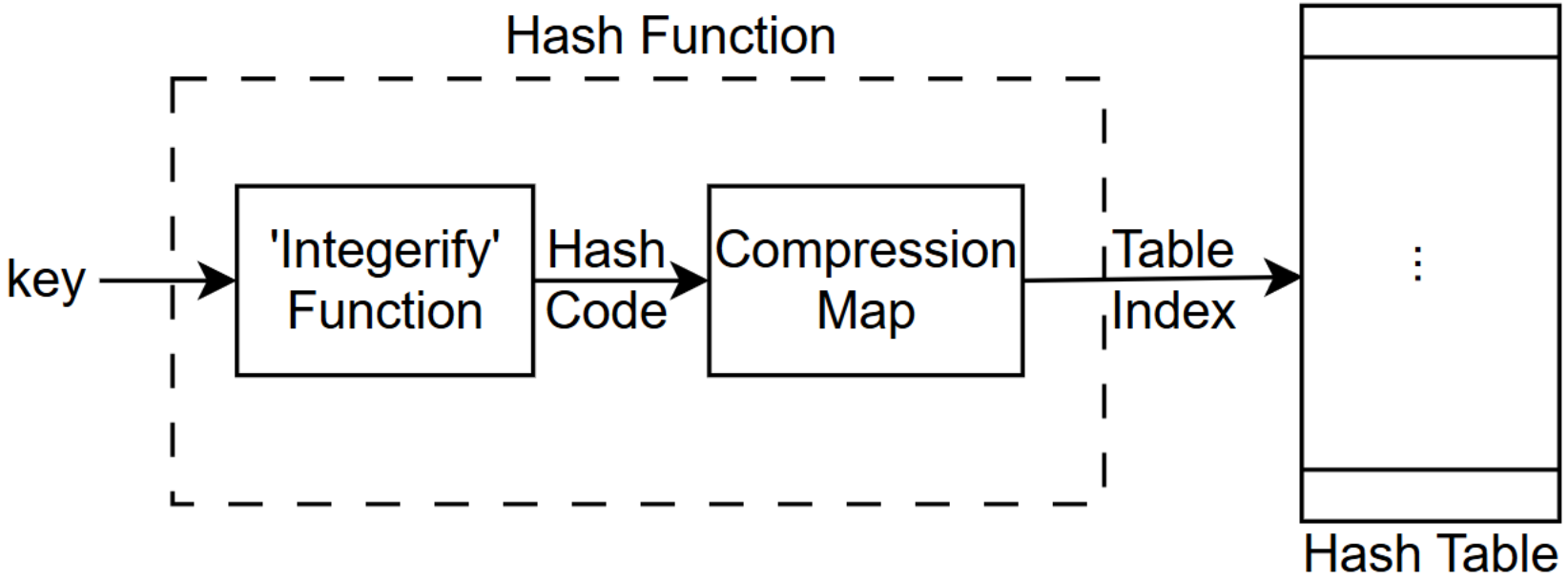
N=	2^5	2^{10}	2^{20}	2^{30}	2^{40}
$\log_2 n$	5	10	20	30	40
$\log_2(\log_2 n)$	2.3	3.3	4.3	4.9	5.3

Introduction to Hashing

- Ideally, what would we want our search big-O to always be?
- O(1)
- …which we cannot achieve in all cases, but that doesn't mean that we shouldn't try! A way that we've tried to achieve this is via *hashing*

Introduction to Hashing

- A general view of hashing:
- ‘Integerify’ Function: needed when key isn’t already an integer
- Compression Map: brings hash code into the table’s index range
- The hashcode() method in Java’s Object class is an ‘Integerify’ function



Introduction to Hashing

- The goal of a hash function is to produce well-distributed table indices (like a random number generator, but reproducible)
- There are three classic hash function techniques we'll be covering:

1. Division Method
2. Multiplication Method
3. The 'MAD' Method (Multiplication, Addition, Division)

Introduction to Hashing

- The Division Method:
- Assuming that the hash table has *m* slots:
- $h(key) = key \,\%\, m$
- Considerations when selecting the table size (*m*):
- Available main memory
- Load factor, aka, $\frac{\#\ of\ keys}{table\ size}$ a ratio that, as it approaches 1, causes "collisions"
- Data patterns, such as choosing *m* to be relatively prime to data (consider a bunch of even-numbered data with an even-numbered *m*)

Introduction to Hashing

- The Multiplication Method:
- Why not just stick with division? Divisions (%) are relatively slow to compute. So, why not use a similar operation (multiplication) that is more efficient?
- $h(key) = \lfloor m * (key * A - \lfloor key * A \rfloor) \rfloor$, where m is the hash table size, and A is a constant such that 0 < A < 1.
- This hash function works like this: $key * A - \lfloor key * A \rfloor$ will create a value [0, 1). Multiplying that by m expands that value to the range of the hash table's indices.

Introduction to Hashing

- The ‘MAD’ Method:
- Addresses another issue with the Division Method: we don’t know that m will be a good choice to randomize the indices for the Division Method. So, why not mix up the keys a bit first?
- $h(key) = |a * key + b| \ \% \ m$, where a, b ∈ Z^+ and m and a are relatively prime
- This has the same form as a basic linear congruential function used to produce pseudorandom numbers: $X_{n+1} = (a * X_n + c) \ \% \ m$

Collision Resolution Strategies

- A collision occurs when two key values hash to the same table index. There are two primary ways of dealing with this:

1. Open Addressing
2. Chaining

Open Addressing

- Open Addressing: Place the key in the ‘next’ available location within the hash table
- The meaning of ‘next’ depends on the flavor of open addressing used; we’ll discuss a few (linear/quadratic probing, and double hashing)
- The table is treated as circular for the purposes of open addressing; if ‘next’ is off the bottom of the table, just wrap back around to the top

Linear Probing

- The ‘next’ location is just the next larger index (wrapping around if necessary)
- All we need to do to implement this is to ‘wrap’ the hash function with another function:
- $lp(key, i) = (h(key) + i)\ \%\ m$
- We parameterize *i* to make walking down the table by any number of indices easy to express

Linear Probing

- Linear probing general function: $lp(key, i) = (h(key) + i)\ \%\ m$
- Let's do a specific example. Let $h(key) = key\ \%\ 7$. What would *lp()* look like?
- $lp(key, i) = (key\ \%\ 7 + i)\ \%\ 7$
- Insert 13, 25, 34, 5, and 19 into the hash table.

0	1	2	3	4	5	6
34	19			25	5	13

lp(13, 0) = 6
lp(25, 0) = 4
lp(34, 0) = 6 (collision!)
lp(34, 1) = 0
lp(5, 0) = 5

lp(19, 0) = 5 (collision!)
lp(19, 1) = 6 (collision!)
lp(19, 2) = 0 (collision!)
lp(19, 3) = 1 (finally!)
Looks awfully like linear search…

Linear Probing

- Linear probing general function: $lp(key, i) = (h(key) + i)\ \%\ m$
- Let's do a specific example. Let $h(key) = key\ \%\ 7$
- $lp(key, i) = (key\ \%\ 7 + i)\ \%\ 7$
- Searching: Find 34. Then Find 42.

0	1	2	3	4	5	6
34	19			25	5	13

lp(34, 0) = 6, but array[6] = 13, not 34. Have to move on!
lp(34, 1) = 0 Success!
lp(42, 0) = 0, but array[0] = 34, not 42. Move on!
lp(42, 1) = 1, but array[1]=19, not 42. Move on!
lp(42, 2) = 2, which is unused, which means 42 is not in the table

Linear Probing

- There are a few issues with linear probing:

1. Primary clustering: the contiguous ‘blob’ of data that tends to form in one region of the hash table during linear probing, which leads to linear searching.
2. Deletion requires special handling. Suppose we deleted 13.

0	1	2	3	4	5	6
34	19			25	5	~~13~~

- Now we try to find 34 or 19, which results in failure! To deal with this, we leave a ‘breadcrumb’ in [6] to continue 34’s search. This is generally known as ‘lazy deletion’ with a ‘tombstone’

Quadratic Probing

- The idea is avoid primary clustering by ‘jumping ahead instead of just ‘walking’
- The adjustment: we need a new wrapper function around our base hashing function
- $qp(key, i) = (h(key) + c * i + d * i^2) \% m$, where c, d ∈ Z^+ and d != 0 (otherwise it would just be a linear hashing variant)

Quadratic Probing

- Quadratic probing general function: $qp(key, i) = (h(key) + c * i + d * i^2) \% m$
- Let's do a specific example. Let $h(key) = key \% 7$, c = 3, d = 2
- $qp(key, i) = (key \% 7 + 3 * i + 2 * i^2) \% 7$
- Let's insert 7, 14, 21

0	1	2	3	4	5	6
7		21			14	

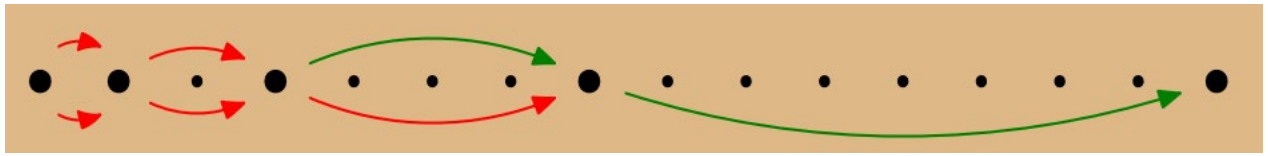
qp(7, 0) = 0
qp(14, 0) = 0 (collision!)
qp(14, 1) = 5
qp(21, 0) = 0 (collision!)
qp(21, 1) = 5 (collision!)

qp(21, 2) = 0 (collision!)
qp(21, 3) = 0 (collision!)
qp(21, 4) = 2

Quadratic Probing

- Quadratic Probing has its own issues…

1. Secondary clustering: just linear probing's primary clustering problem, but in a quadratic sense



2. It is possible that an existing open slot may never be reached, depending on the data, *c*, and *d*
3. Memory locality: we may jump to distant parts of the table that are not currently paged into RAM or the CPU's cache(s)

Double Hashing

- Tired of all these clustering issues? Me too!
- Double Hashing tries to avoid them by: basing our hashing on a pair of hash functions
- $dh(key, i) = (h_1(key) + i * h_2(key)) \% m$
- To avoid similar hashing behaviors, h_1 and h_2 should be different types of hash functions

Double Hashing

- Double hashing general function: $dh(key, i) = (h_1(key) + i * h_2(key)) \% m$
- Let's do a specific example. Let $h_1(key) = \lfloor 7 * (key * .5 - \lfloor key * .5 \rfloor) \rfloor$ and $h_2(key) = |3 * key + 3| \% 7$
- Let's insert 3, 4, 5, 6

0	1	2	3	4	5	6
4			3	5		

dh(3, 0) = 3
dh(4, 0) = 0
dh(5, 0) = 3 (collision!)
dh(5, 1) = 0 (collision!)
dh(5, 2) = 4

dh(6, 0) = 0 (collision!)
dh(6, 1) = 0 (collision!)
dh(6, 2) = 0 (collision!)
dh(6, 3) = 0 (collision!)
… Never finds an open spot!

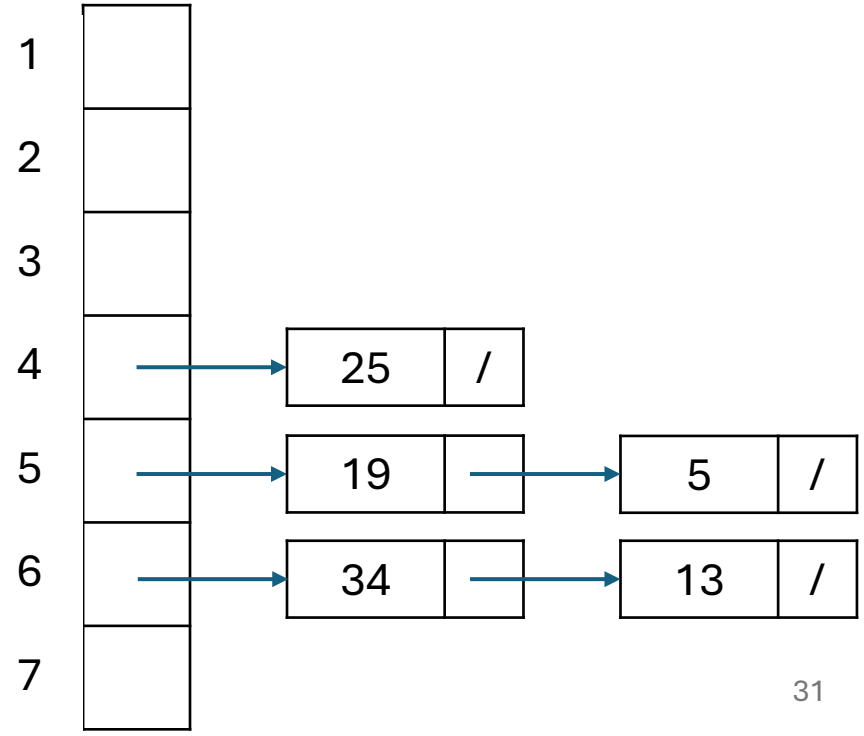
Double Hashing

- Unfortunately, double hashing still has issues:

1. Memory locality (page and cache misses) are still an issue, just like quadratic probing
2. Can still potentially miss open slots (again, like quadratic probing)
3. And our hashing calculations become a little more time-consuming!

Chaining

- Chaining: Turn the hash table entries into head pointers to linked lists of keys.
- For example, *m* = 7, h(key, i) = (key % i) % m, and our task is to insert: 13, 25, 34, 5, and 19
- h(13, 0) = 6
- h(25, 0) = 4
- h(34, 0) = 6
- h(5, 0) = 5
- h(19, 0) = 5
- The chains isolate the collisions
- We also don't need *i*! Just prepend to the LLs



Chaining

- Definition: *Load Factor*
- *The Load Factor of a hash table is the ratio of the number of keys (n) to the number of table slots (m)*

- The average chain length is the load factor: $O(\frac{n}{m})$
- The average search is $O(\frac{n}{m} + c)$ (c is for the array dereferencing)
- Assuming that n ∈ O(m):
- Our expected search time: $\frac{n}{m} \Rightarrow \frac{O(m)}{m} \Rightarrow O(1)$
- The worst case is O(n), when all the keys are in the same chain

Final Comments on Hashing

- First, is there a way to declutter a cluttered hash table?
- Yes! Move all the data into a new, larger hash table. When is it a good idea to move up? It depends on the context, but in general…
- For Open Addressing, when the load factor is > ~75%.
- For Chaining, when the load factor is > ~300%, or when the longest chain exceeds some defined threshold.
- How large should the new table be?
- “Double the old” is a common heuristic (this is what Java’s HashMap does!)
- Note that rehashing isn’t cheap!

Final Comments on Hashing

- There’s a lot more to the hashing story. A few examples:

1. Universal Hashing: Creates a new hash function for each hash table, to balance performance and foil adversaries
2. Perfect Hashing: Guarantees a worst case O(1)! Usually requires knowing your data in advance
3. Bloom Filters: Applies hashing to secondary storage, guaranteeing no false negatives for search queries, and a manageable number of false positives

Summary of Searches

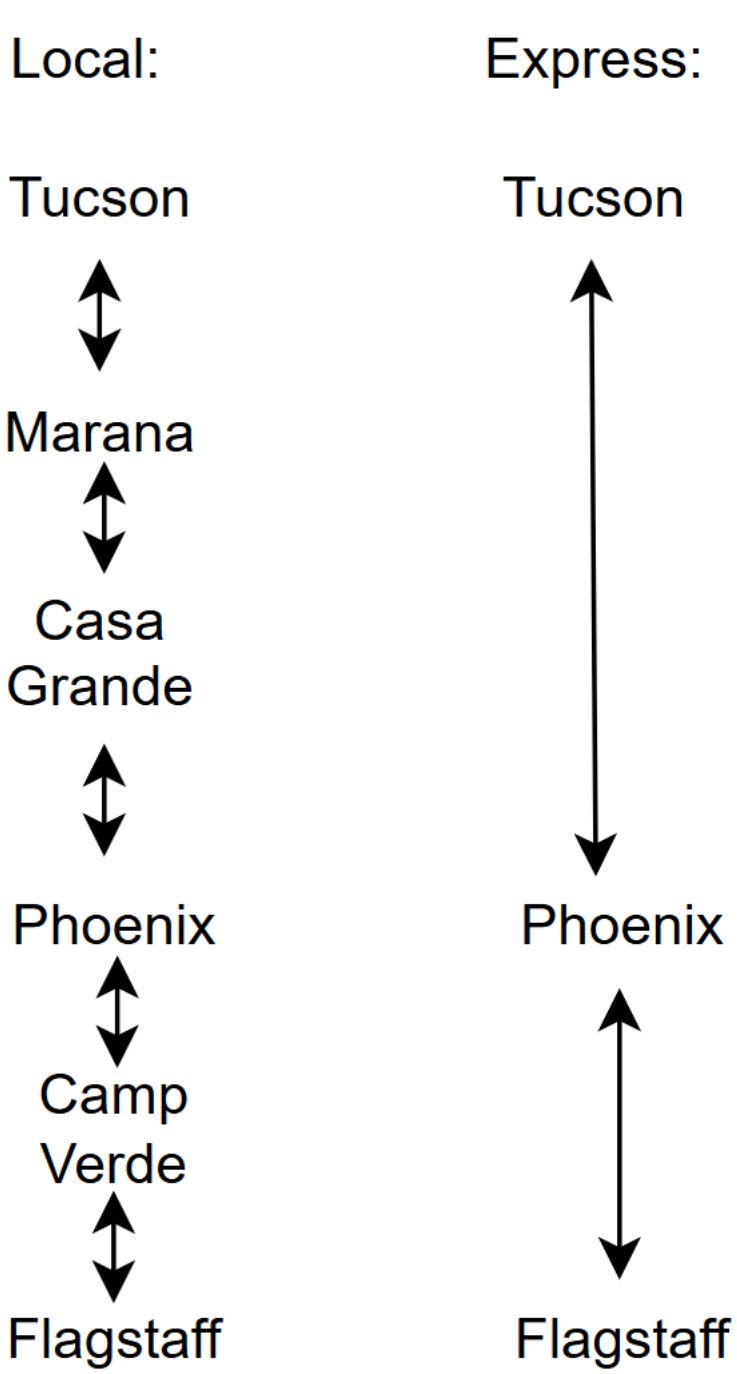
- So far, we've discussed:
- Linear (Sequential) Search
- Binary and Interpolation Search
- Hashing
- All are worst-case O(n), except…
- Binary Search! But Binary Search requires directly-accessible, pre-sorted data. Is it possible to get as good a worst-case when searching a presorted linked list?

Skiplists

- Skiplists are an example of a *probabilistic* data structure
- This means that the construction of skiplist structure is (partially) determined by chance!
- Another advantage of skiplists is that it’s a form of linked list, which means it has a significant advantage over arrays: we can insert data without shifting the locations of existing data

Skiplists

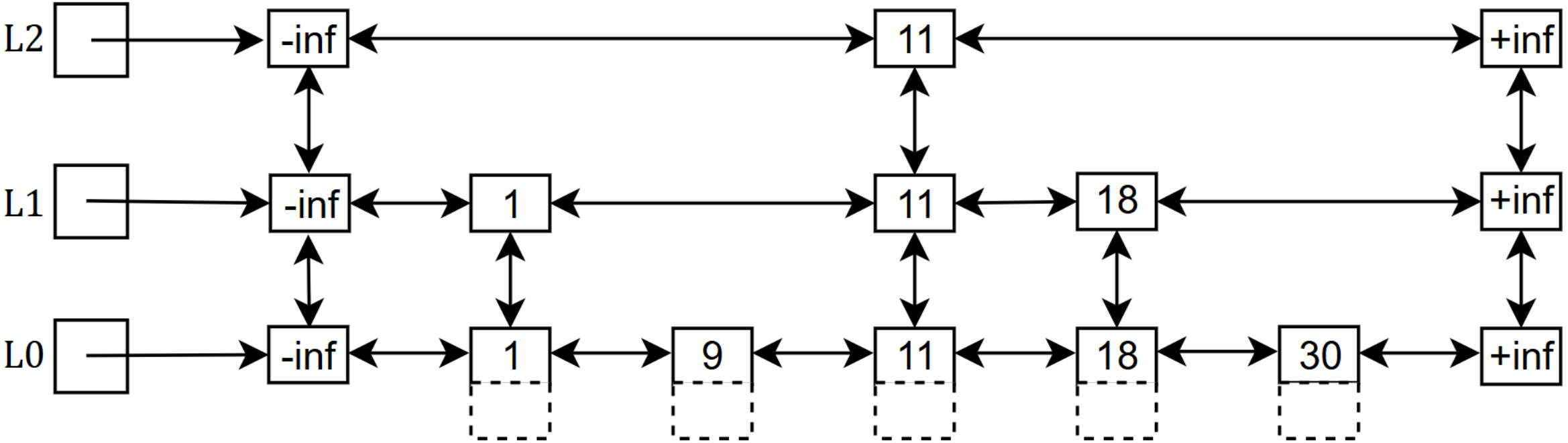
- Here's the basic concept: imagine you're planning a bus route to Camp Verde. There are two available routes from Tucson, Local and Express.
- What would the most efficient route be for your trip?
- Take the Express to Phoenix, and the Local to Camp Verde!



Skiplists

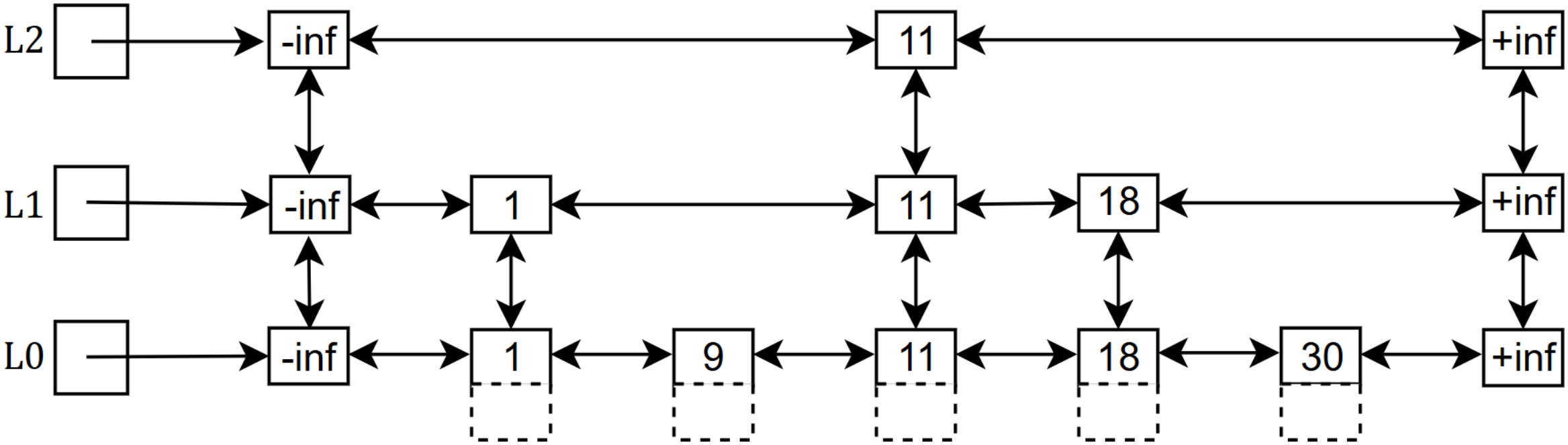
- Skiplists pick the “bus stops” for a route randomly—which would be a big problem normally—but we’ll make it work by having multiple levels of “express bus routes.”
- With this “express route” model in mind, let’s look at an example of a skiplist

Skiplists



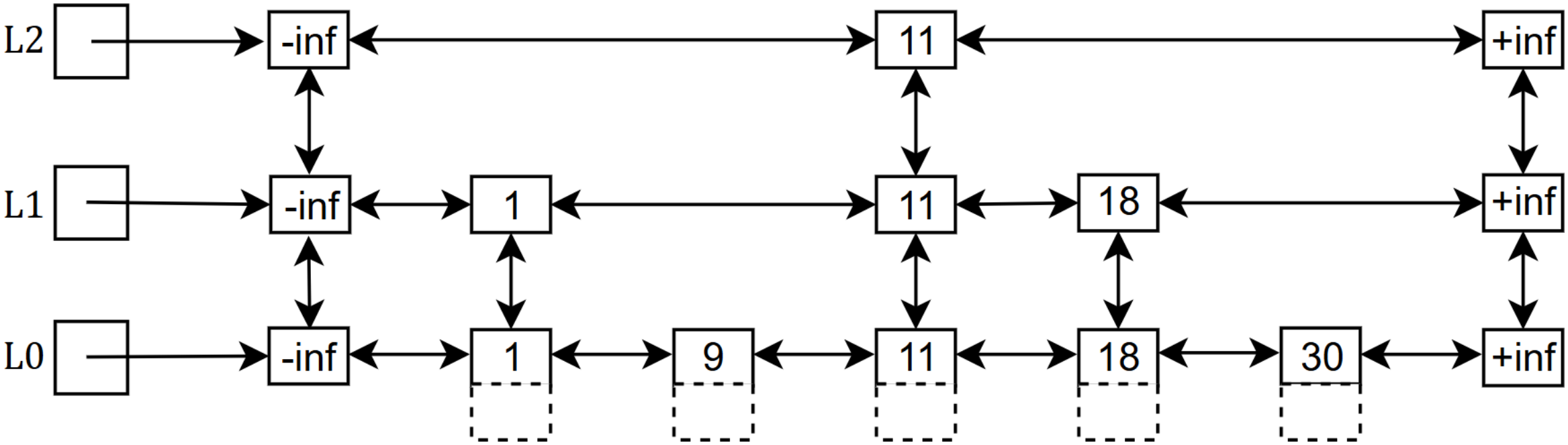
- L0 has all of the keys, in order (the “local” bus stops)
- Dashed boxes represent each key’s associated data
- Vertical stacks of keys and copies are called *towers*
- Implementations usually skip the +/- infinity towers

Skiplists



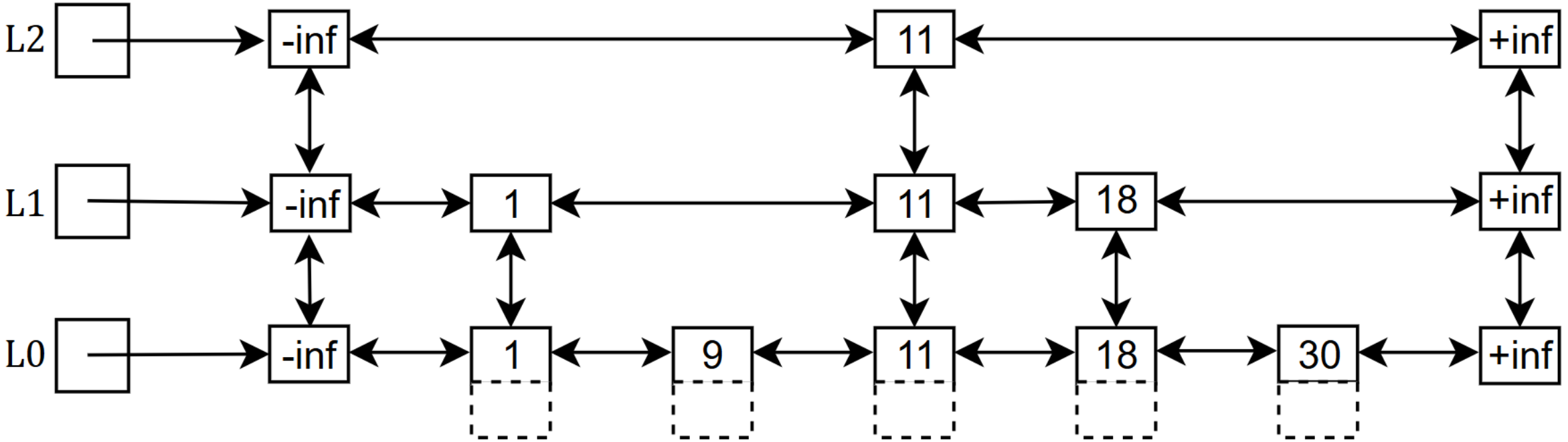
- L_i contains ~1/2 of the keys of L_{i-1}, which means…
- A skiplist has $O(\log_2 n)$ levels (which is the same as well-balanced BST)
- All of the lists (vertical and horizontal) are doubly-linked lists

Skiplists



- Let's try a search. Our target is 30. Here's our search process:
- Search L2 to 'bracket' the target (between 11 and +inf)
- Drop down 11's tower to L1. 'Bracket' between 18 and +inf.
- Drop down 18's tower to L0, sequential search to the right. Success!

Skiplists



- Let's try another search. Our target is 0. Here's our search process:
- Search L2 to 'bracket' the target (between -inf and 11)
- Drop down -inf's tower to L1. 'Bracket' between -inf and 1.
- Drop down -inf's tower to L0, sequential search to the right. Failure!

Skiplists

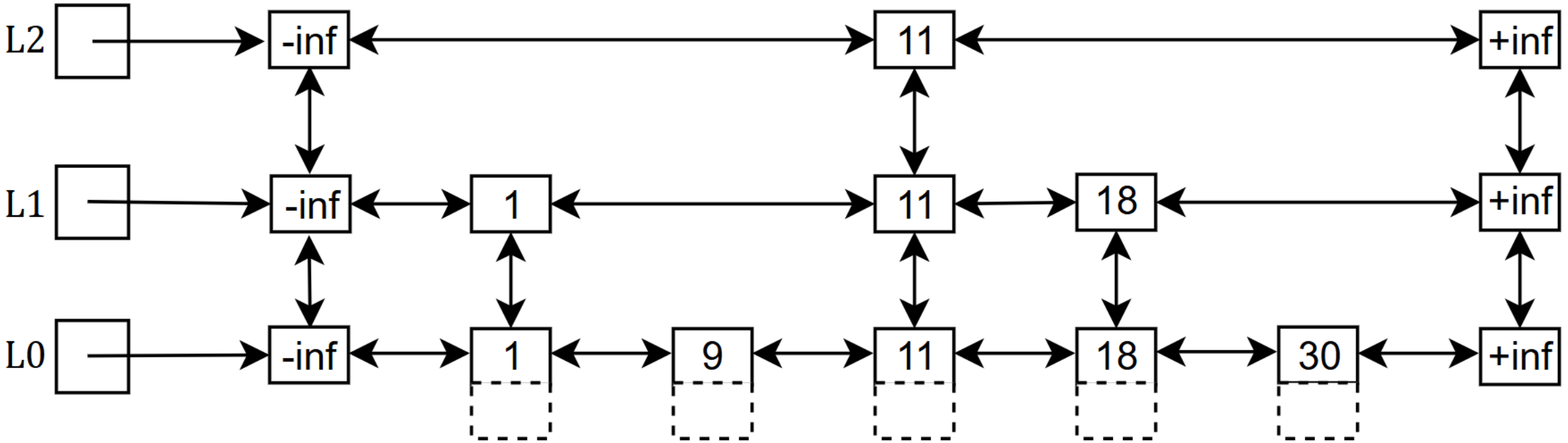
- Here's the pseudocode for the search process:

Skiplist Search:
Given a search target,
i ← index of the top list
Do:
    sequentially search list L_i for the key whose subsequent key
        is greater than the search target
    i ← i – 1
While i > 0;
Sequentially search list L0 for the target

Skiplists

- Skiplist search analysis:
- We need to search *parts* of $O(\log_2 n)$ levels. In other words, we travel horizontally on each level for *some* distance… but how far?
- Answer: about 2 nodes. Why? We expect that every other value on each level was also on the higher level, which bounds our search on the current level.
- So, the total cost will be the number of "drop-downs" plus the constant search time per level
- $cost = \log_2 n + (\log_2 n + 1) * c = (c + 1)\log_2 n + c$
- Which means we *expect* each search to be $O(\log_2 n)$

Skiplists



- Now let's try an insertion. Insert 14. We need to insert where we would expect to find it, so…
- Search for 14 (record where the search fails)
- Insert 14's node in L0 (between 11 and 18). Done! Except…

Skiplists

- For every inserted node, we have to grow its tower. How far?

Skiplist Insertion:
Given a key to be inserted,
Search for the key, remembering failure position
Flip a (virtual) coin <= NUM-LEVELS times, counting the quantity of heads found before the first tail is seen
Build a tower of count+1 levels at the failure position

Skiplists

Skiplist Insertion:
Given a key to be inserted,
Search for the key, remembering failure position
Flip a (virtual) coin <= NUM-LEVELS times, counting the quantity of heads found before the first tail is seen
Build a tower of count+1 levels at the failure position

- Note: if NUM-LEVELS = 3, we can only build a tower of 1-4 levels. An alternative for this approach is to ignore the upper limit.
- Analysis: Search ($O(\log_2 n)$) plus tower-building ($O(\log_2 n)$), gives an *expected* $O(\log_2 n)$
- Why don't we deterministically pick tower locations and height? Because the data might change.

Skiplists

- Skiplist deletion process:

1. Search for the victim
2. If found, travel up the tower, deleting nodes as we go

- Analysis of deletion: $O(\log_2 n)$, again *expected*

Skiplists

- Skiplist space requirements:
- L0 has all of the keys: O(n) nodes
- L1 has an expected ~n/2 nodes
- L2 has an expected ~n/4 nodes
- L3 has an expected ~n/8 nodes… etc.
- $\sum_{i=1}^{x} \frac{1}{2^i}$ approaches 1. Thus, $\sum_{i=1}^{x} \frac{n}{2^i}$ approaches n
- So O(n) nodes for L0, and O(n) nodes for the rest… just O(n) space needed!

Interpolation Search Practice

- Search for 59 in the array below:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
10	18	22	28	34	36	48	59	62	64	66	81	85	89	90

- probe = $low + \left\lfloor \frac{target - list[low]}{list[hi] - list[low]} * (high - low) \right\rfloor$
- $0 + \left\lfloor \frac{59-10}{90-10} * (14 - 0) \right\rfloor = 8$ First probe will be 8. 62 > 59, so…
- New high is 7, low remains at 0.
- $0 + \left\lfloor \frac{59-10}{59-10} * (7 - 0) \right\rfloor = 7$ Second probe will at 7. Success!